COSC460

Computer Science Honours Project

University of Canterbury

A Comparison of Worst Case Performance of Priority Queues used in Dijkstra's Shortest Path Algorithm

By Alex Vickers

Department of Computer Science

University of Canterbury

1986

Supervisor : Dr. Alistair Moffat

ABSTRACT

This report presents results of experiments comparing the worst case performance of Dijkstra's Single Source Shortest Path algorithm using different priority queues. A description and worst case analysis are given of the five priority queues which were considered.These were an array of keys, array of pointers, binary heap, alpha heap and Fibonacci heap. To produce worst case performance of Dijkstra's algorithm it was necessary for the author to design a method for generating graphs which would induce this behaviour. A family of graphs with this property was discovered and their description is given within. Results of applying Dijkstra's algorithm to graphs of this type, using each of the priority queues are listed in the appendix while the more interesting outcomes are graphed and analysed in the main body of the report.

CONTENTS

Section **Page**

1. Project Description

The purpose of this project is to determine which of a number of different data structures give the best performance when used to implement priority queues and used in particular by graph algorithms of which Dijkstra's Shortest Path Algorithm is highly typical. Of special interest is the recently invented priority queue known as a Fibonacci Heap, [FredTarj84, 338 - 346] or F-Heap for short which theory shows should be better than other known priority queues under certain conditions.

The project involved making some initial decisions about which data structures were to be compared, what machine and what language were to be used. Then the data structures chosen were implemented and Dijkstra's Shortest Path Algorithm was run using priority queues implemented using each of the chosen data structures on appropriate graphs (see section 4). Since part of the project was to determine whether F-heaps could be shown in practice to be better than other known priority queues it was necessary to decide under what conditions the F-heap would give the fastest times, then design and implement an algorithm for generating graphs which would cause these conditions to arise.

2. Graph Terminology and the Order Notation

A graph is an abstract concept consisting of some number n, of nodes connected by some number m, of edges. A connected graph is a graph where every node can be reached by following a sequence of edges, called a path, beginning at any other node. If a graph is not connected then it consists of at least two subgraphs such that there is no path from a node in one subgraph to another, these subgraphs are called components. Associated with each edge there is a number called the cost of the edge, which for example in the real world might represent the cost of petrol in travelling between two cities or the time for some system to change between two states. The cost of a path is the sum of the costs of the edges on that path. The distance between two nodes is the minimum cost of traversing the graph from one node to the other i.e. the minimum cost over all possible paths between the nodes, which can be important to know if one wishes to save money or time. Dijkstra's Shortest Path algorithm finds the distance from some specified source node to the other nodes in a graph with non-negative edge costs. In this project the author has only considered connected graphs with non-negative edge costs since it is desired that Dijkstra's algorithm can be applied to them and there is little point in considering non-connected graphs since applying Dijkstra's algorithm to such a graph is equivalent to applying it to just the component of the graph containing the source node.

A special case of a graph is a tree which is a graph with no cycles, a cycle being a path of more than one edge which connects a node to itself. Trees were used as priority queues for Dijkstra's algorithm. Usually a tree is said to consist of nodes joined by edges or branches however within this report the nodes of a tree will be called records in order to avoid confusion between the nodes of the graph, to which Dijkstra's algorithm is applied and the nodes of a tree, used as a priority queue.

Some priority queues perform better than others under certain conditions. The order notation is an analytical measure of the efficiency of a priority queue, or indeed any algorithm. If an algorithm takes f(n) seconds to work then it is of the order g(n), written O(g(n)), if for some constant k and sufficiently large n, $f(n) \leq k * g(n)$. For example, $f(n) = 3*n^2*\log n + 7*n = O(n^2 \log n)$. The order notation can also be used to measure the amount of memory an algorithm uses, however this is not important for the priority queues being considered which all require some amount of memory space proportional to n, the number of nodes in the graph, and normally memory is a resource in far greater abundance than time.

3. Priority Queues

Generally any priority queue needs to provide functions for the operations insert, deletemin and delete. It should be possible to consider a priority queue as a 'black box' to which objects can be given each with some key value. The priority queue should then keep its collection of objects in some form and provide an interface for requesting the deletion of the object of minimum key, or of some arbitrary object, and for the return of the information required. The operation decreasekey can be implemented by using delete and insert to delete the node and then insert it with its new key however this is likely to be inefficient and so a further function called decreasekey is provided. Dijkstra's algorithm requires a priority queue to provide the functions insert, deletemin and decreasekey thus these will be described in the next sections for the five priority queues considered.

3.1 Array of Keys

An array a, of keys is used such that a[i] is the key value of the node i. The absence of a key is represented by a unique value. This is rem in figure 3.1. This was implemented as a Pascal array of integers with a large integer used for ∞ and maxint used for rem.

Nodes :	1	2	3	4	5	6	7	8	9	10	11
Keys :	3	5	2	∞	8	9	rem	10	1	∞	rem

Figure 3.1 An array of keys for 11 nodes, rem indicates that the node has been removed.

3.1.1 Insertion in an Array of Keys.

To insert the key for a node into the array, the key is assigned to that cell of the array i.e. a[node] ← key.

3.1.2 Deletemin in an Array of Keys.

The array is scanned once to find the node of minimum key, ignoring those entries with the key value rem, the minimum node is saved and its key value in a is replaced by rem to show that it is no longer there.

3.1.3 Decreasekey in an Array of Keys.

The key of the node to be decreased is assigned its new value, a[node] $\leftarrow$ newkey.

3.1.4 Worst Case Time Analysis of an Array of Keys.

Insertion requires O(1) time since the node number is used as an index to the appropriate record. Deletemin requires O(n) time because each of the n entries must be looked at to find the one of minimum key. Decreasekey however requires just O(1) time since the value to be changed can be immediately indexed to.

3.2 Array of Pointers to Keys

An array ar, of pointers to records is used. Each record contains a node number and its corresponding key. A separate index array is kept so that the record of a given node can be found immediately. A count of the number of nodes still represented in the data structure is maintained and the structure is kept such that the pointers with indices from 1 upto this count reference records of nodes still in the structure.

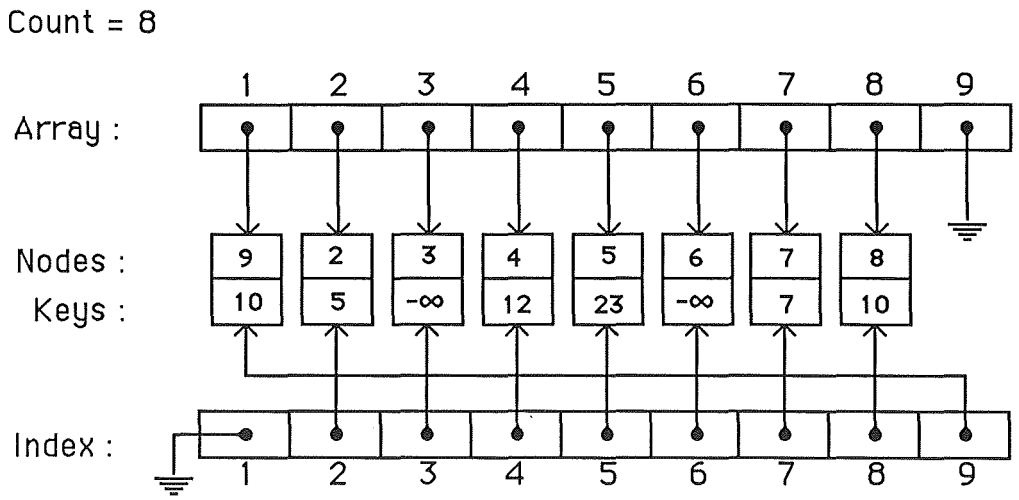


Figure 3.2 An array of pointers to 9 records, one of which (node 1) has been deleted leaving a nil index pointer.

3.2.1 Insertion in an Array of Pointers.

A node and its key are copied into a record pointed to from the next available cell of the array, its index found from the incremented count of the number of items in the structure (initially 0) i.e. the record is pointed to by ar[count + 1].

3.2.2 Deletemin in an Array of Pointers.

The array is scanned up to the current node count and the node of minimum key among those referenced is found. The node is then removed and the corresponding pointer is made to point to the node pointed to by the pointer contained in ar[count] before count is decremented.

3.2.3 Decreasekey in an Array of Pointers.

The record for the node whose key is to be decreased is found immediately from the index array. The key is then just assigned its new value.

3.2.4 Worst Case Time Analysis of an Array of Pointers

Insertion of a record takes O(1) time since the incremented count is used as an index to the array cell which should point to the record. Deleting the minimum record after all n records have been inserted first involves scanning n entries, then for the next deletemin n-1 entries must be scanned, then n-2 and so on. Thus the deletemin function for an array of pointers priority queue actually takes $n + n-1 + n-2 + ... + 2 + 1 = (n^2-n) / 2$ operations, half the number of an array of keys, and so should be better when n is large enough to drown the extra overhead. However both array methods take $O(n^2)$ time for deletemin. Decreasekey in an array of pointers is accomplished in O(1) time because the appropriate record can be found immediately from the index array and then its key can be assigned the new value.

3.3 Binary Heap

A binary heap is conceptually a tree within which every record has at most two branches and contains a key value together possibly with some associated information such as the number of the graph node for which the key is the currently known distance. One further condition for such a tree to be a binary heap is that the key of a record must be less than or equal to each of the keys of the records which are contained in the left and right subtrees from that record.

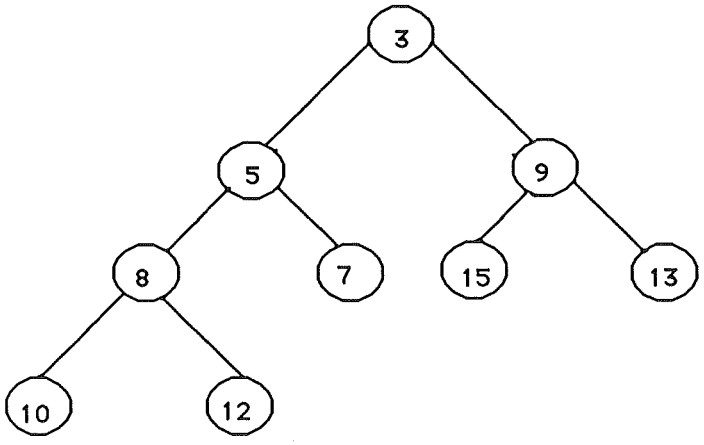


Figure 3.3 An example of a binary heap.

In the example of a binary heap in figure 3.3 each record is labeled with its key value. It can be seen that the top record's key, 3 is less than 5 and 9 which are less than 8 and 7, 15 and 13 respectively. As a consequence of this heap order the top record of the heap contains the key of minimum value over the entire heap.

The binary heap was represented in a Pascal array so that the parent of a record at cell i of the array would be at cell $\lfloor i / 2 \rfloor$ of the array and so the left and right children of a record would be at the cells with indices 2*i and 2*i + 1 respectively. A further explanation of this can be found in [HoroSahn84, 349 - 350]. The other details of the implementation proceeded basically as described below based on this representation. The three basic operations which need to be performed on the binary heap are described below.

3.3.1 Insertion in a Binary Heap

To insert a record with a specified key into a binary heap involves firstly attaching another record to the bottom of the tree (this is done in breadth first fashion so that the depth of the tree only increases when it is unavoidable) and then to ensure that the heap order is

maintained the content of the record must be continually swapped with that of its parent until the key of the inserted record is greater than or equal to that of its parent or it is the top record. If a record with key value 4 is inserted into the binary heap of figure 3.3 then the heap in figure 3.4 results.

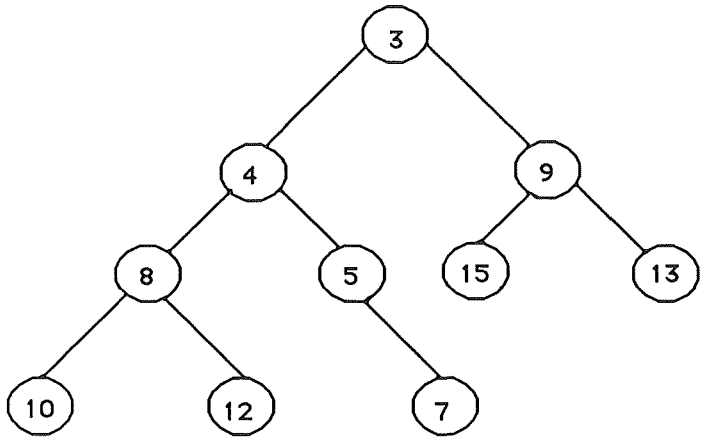


Figure 3.4 The heap after the record with key 4 has been inserted.

The record was attached below the record previously labelled 7 and then the contents of this record and its parent's were swapped, putting the 7 in its final position. The key of the record labelled 5 above was then 4 and this was less than 5, the record of its parent, so these were then swapped and no more swapping was required because the key 3, of its parent was less than its key. It can be seen that these operations leave the structure as a binary heap again.

3.3.2 Deletemin in a Binary Heap

Deleting the record of minimum key value involves firstly copying the contents of the top record and then replacing this record with the rightmost record at the bottom level. This record is then swapped with its child of minimum key down the tree until the heap property is regained i.e. until it reaches a position where its children have keys greater than or equal to it or it has no children. If for example a deletemin is performed on the binary heap of figure 3.4 then the binary heap of figure 3.5 would result.

In this example the top record was deleted and replaced with the record of key 4 which was in turn replaced by the record with key 5 and the record which previously contained the key 5 was removed.

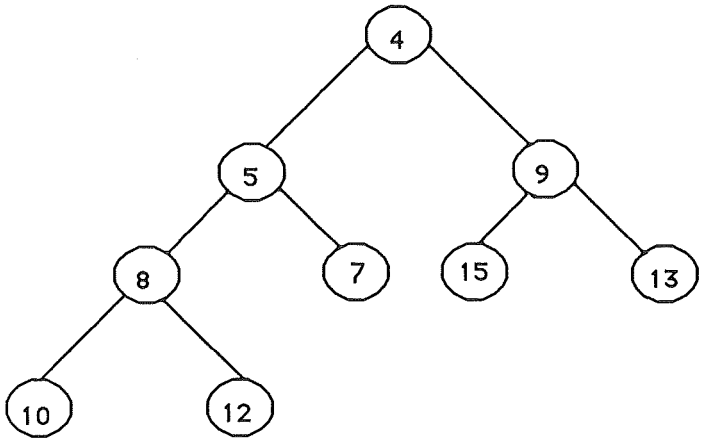


Figure 3.5 The heap after the minimum record has been deleted.

3.3.3 Decreasekey in a Binary Heap

When the key of a record is decreased the heap order within the subtree rooted at that record is preserved but the heap order of the rest of the heap will be disrupted if the key of the record decreases to less than the key of its parent. So after decreasing the key of a record the record is swapped up the tree until its key is greater than or equal to that of its parent. In the figure 3.6 the heap of figure 3.5 has had the key of the record previously labelled 8 decreased to 1.

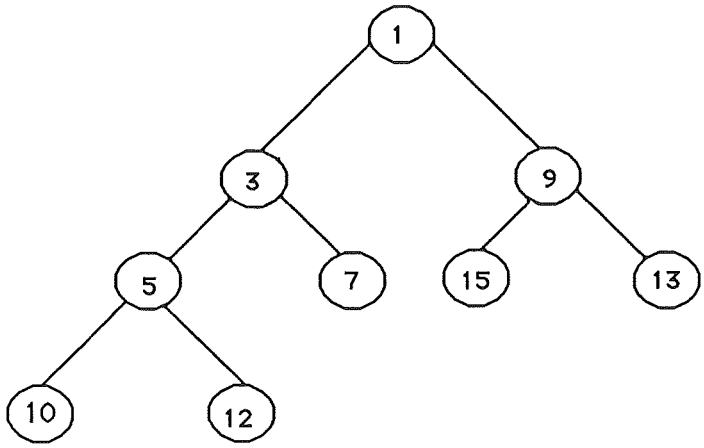


Figure 3.6 The heap after the 8 record is decreased to 1.

3.3.4 Worst Case Time Analysis of a Binary Heap

Insertion requires O(log n) time in the worst case since an inserted record may have to be swapped up from the bottom to the top of the tree. Deletemin requires O(log n) time in the worst case since it could be necessary to swap a record from the top down to the bottom level i.e. the full depth of the tree which for a binary tree of n records is $\log_2 n$ = O(log n). Decreasekey also requires O(log n) time in the worst case where a record at the bottom level has its key decreased to a value below that of the top record and hence is swapped up the full depth of the tree.

3.4 Alpha Heap

The alpha heap, like a binary heap is a heap ordered tree except that an alpha heap does not necessarily have just two branches (or less) from each record. The maximum number of branches from each record in an alpha heap is called alpha. As with the binary heap the alpha heap was implemented using a Pascal array. The insert, deletemin and decreasekey operations are all carried out in the same way as described for a binary heap.

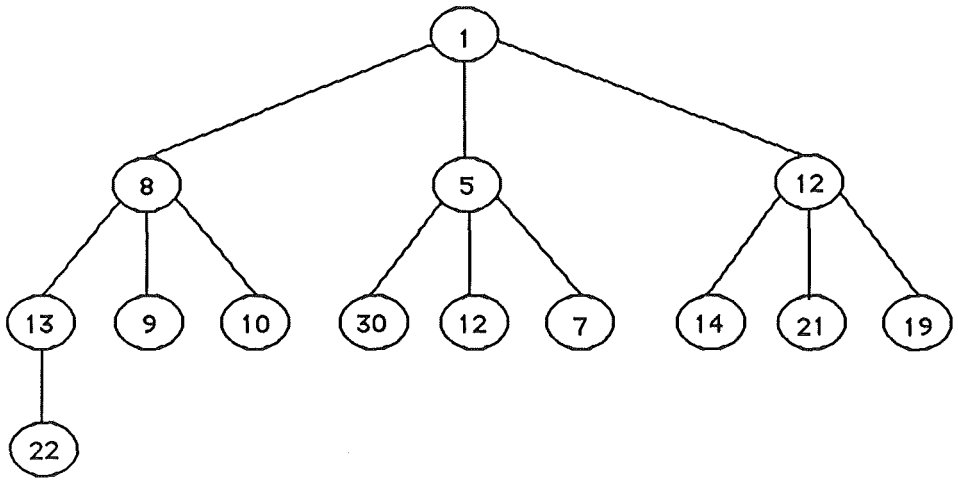


Figure 3.7 An Alpha Heap for a Graph with Edges/Nodes = 3.

3.4.2 Worst Case Time Analysis of an Alpha Heap

Insertion requires $O(\log_{\alpha} n)$ time in the worst case since an inserted record may have to be swapped up from the bottom to the top of the tree. Deletemin requires $O(\alpha \log_{\alpha} n)$ time in the worst case since it could be necessary to swap a record from the top down to the very bottom level i.e. the full depth of the tree which for an alpha tree is $O(\log_{\alpha} n)$. Decreasekey requires $O(\log_{\alpha} n)$ time in the worst case where a record at the bottom level has its key decreased to a value less than the previous minimum key and hence is swapped up the full depth of the tree.

3.5 Fibonacci Heap

A record within this structure is contained within a doubly linked list of one or more records all of which have the same parent i.e. they are siblings. Each record has a pointer to one of its children from which its other children may be found by following the doubly linked list of siblings. As well, each record with a parent has a pointer to that parent. The record in the F-heap of minimum key is pointed to by a pointer called the min pointer through which the entire F-heap is accessed. If the min pointer is nil then the F-heap is empty. The F-heap is heap ordered which means that the key of every record is less than or equal to each of the keys of its descendants. For reasons which will be explained soon, each record also requires a mark field. Thus fields for the following are required in each record :

1. The key.
2. Any associated information.
3. A pointer to its left sibling.
4. A pointer to its right sibling.
5. A pointer to one of its children.
6. A pointer to its parent.
7. Its rank i.e. a count of the number of children it has.
8. A mark field.

In order that the record for any given node can be accessed immediately an array, 'find' of pointers to nodes indexed by their number is kept so that for example find[3] would point to the record in the F-heap for node 3 of the graph while the node exists in the F-heap.

3.5.1 Insertion in a Fibonacci Heap

To insert a new record into the heap h, a new heap containing just the record to be inserted is melded with h. To meld two heaps h_1 and h_2, their root lists are combined into a single list and the min pointer of the new heap is made to point to the lower of the two mins from h_1 and h_2.

3.5.2 Deletemin in a Fibonacci Heap

To begin with the minimum record is removed and the list of its children is concatenated with the list of roots of the F-heap. Then while there are any two roots in this list with the same

rank they are linked. Linking involves making the root of greater key the child of the other root. Once there remain no trees of the same rank in the list the min pointer is set to point to the new record of minimum key.

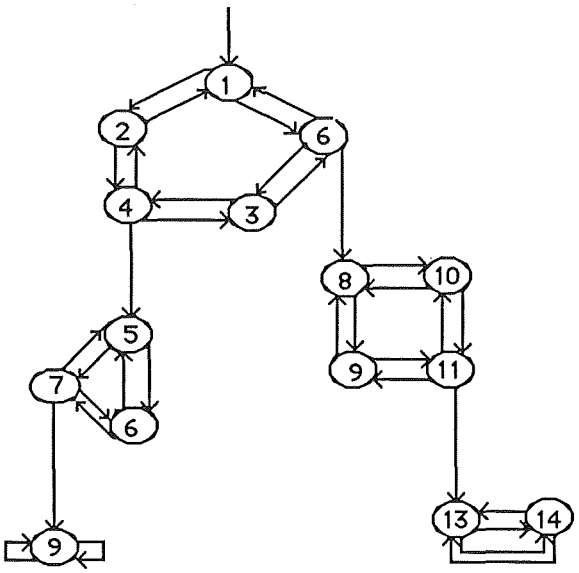


Figure 3.8 A Fibonacci Heap with minimum record of key 1.
To avoid cluttering the diagram, pointers to
parent records and nil child pointers are not shown.

3.5.3 Decreasekey in a Fibonacci Heap

To perform the decreasekey operation for some graph node x, firstly the corresponding record in the F-heap is found from the index array and its key is decreased to the new value. Next in order that the heap order of the F-heap be preserved the record is cut i.e. detached from its siblings and parent and is repositioned in the root list at the top of the F-heap. After inserting this new root record the min pointer will have to be reset if the key was decreased to a value less than that of the previous minimum key. There is a further detail necessary to obtain the desired time bound of O(1) time for this operation. During a deletemin, when a root record is made the child of another record in a linking step, as soon as two of its children are cut by a decreasekey operation it must also be cut. The mark field is used to keep track of the number of children a record has lost due to cuts. During a linking step the child record is unmarked.When a record is cut, if its parent is unmarked then it is marked otherwise if its parent is marked then it too is cut. In this way it is possible for a decreasekey operation to cause a large number of 'cascading' cuts.

3.5.5 Worst Case Time Analysis of a Fibonacci Heap

Insertion requires O(1) time. Deletemin requires O(log n) amortized time in the worst case. Decreasekey requires O(1) amortized time in the worst case. The analyses for these are rather complicated and can be found in [FredTarj84].

3.6 Summary of Worst Case Time Analyses

Figure 3.9 summarises the time bounds found for each of the priority queues discussed above. Clearly the Fibonacci heap exhibits the best time bounds, it being the only priority queue with a bound equal to the best of the others for each of the three operations.

Priority Queue	Insert	Deletemin	Decreasekey
Array	$O(1)$	$O(n^2)$	$O(1)$
Aryptr	$O(1)$	$O(n^2)$	$O(1)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$
Alpha Heap	$O(\log_\alpha n)$	$O(\alpha \log_\alpha n)$	$O(\log_\alpha n)$
Fibonacci Heap	$O(1)$	$O(\log n)$	$O(1)$

Figure 3.9 The time bounds for the five priority queues.

4. Dijkstra's Shortest Path Algorithm

Given a graph with n nodes and m edges each with some associated non-negative cost this algorithm can find the distance from a specified source node to all the other nodes in the graph. The distance from one node to another is the minimum cost of traversing any path between them. Dijkstra's algorithm works in conjunction with a priority queue for storing the currently known distance of each node from the source. These distances are updated by way of the decreasekey operation and once the distance to a node is known it can be deleted by way of the deletemin operation.

As the latter sentence suggests Dijkstra's algorithm finds each node in order of non-decreasing distance from the source. Initially the source is inserted in the priority queue with a key of 0, since it is known that there is a cost of 0 in travelling from the source to itself, and all other nodes are inserted with a cost of ∞. The algorithm then loops n times, in each finding the next furthest node from the source and updating the keys of other nodes. If we denote the cost of the edge (p, q) from node p to node q by cost(p, q), the current key in the priority queue of a node q by distance(q) and the decreasekey operation by decreasekey(q, newkey) meaning that node q has its key decreased to the value newkey then we may write Dijkstra's algorithm in pseudocode as follows:

```
Insert source node with key = 0.
Insert all other nodes with key = ∞.
While the priority queue is not empty :
        Deletemin, giving the node, p and distance, d
        Assertion:  The distance to node p is d.
        For each node, q adjacent to p :
                If d + cost( p, q ) < distance( q ) then
                        decreasekey( q, d + cost( p, q ))
```

Further more detailed explanations of how and why Dijkstra's shortest path algorithm works can be found in [AHU74, 207 - 209] and [AHU83, 203 - 207].

4.1 Worst Case Time

From the above pseudocode it can be seen that for a graph of n nodes and m edges there will be n insert operations, n deletemin operations and in the worst case there will be a total of m decreasekey operations. Thus the worst case time for Dijkstra's algorithm using any given priority queue is:

$$WCT(\text{Priority Queue}) = O(n * \text{insert} + n * \text{deletemin} + m * \text{decreasekey}) \quad [1]$$

where insert stands for the time to perform the insert operation, deletemin stands for the time to perform the deletemin operation and decreasekey stands for the time to perform the decreasekey operation, all in the worst case. Generally by worst case we mean that situation in which the operation will require the maximum possible time.

The behaviour of Dijkstra's algorithm on three particular types of graph were investigated. The first of these was sparse graphs where the number of edges is small compared to the number of nodes i.e. $m = O(n)$. These are of interest simply because large graphs often have few edges per node e.g. communications networks or the roads in a country where each node only has edges to those nodes near it. Dense graphs where $m = O(n^2)$ are of interest simply because they represent the other extreme. No runs were made of Dijkstra's algorithm on dense graphs since the result, that an array-type priority queue performs best, is well known. In between these two extremes there are, as they shall be refered to in this report, middle graphs. A middle graph has an edge to node ratio of the order of the log of the number of nodes i.e. $m = O(n * \log n)$. These are of interest because it is on such a graph that the Fibonacci Heap has an asymptotically superior time bound and hence should be faster than the other priority queues, provided that the number of nodes is sufficiently large.

By substituting the time bounds, summarised in figure 3.9, into equation [1], the worst case time bounds for Dijkstra's algorithm may be found. These are summarised in figure 4.1, below. It can be easily derived that for an alpha heap the best choice of alpha when used in Dijkstra's algorithm is $\alpha = m/n$. The last row of the table in figure 4.1 gives the fastest priority queue, assuming sufficiently large n, for each of the three categories of graph, namely sparse where $m = O(n)$, middle where $m = O(n * \log n)$ and dense where $m = O(n^2)$. It can be seen from this that the F-heap is asymptotically superior to the other priority queues for graphs with $O(n * \log n)$ edges. For the sparse graphs, although the three heap methods take the same order of time, the binary heap has the least overhead and so should be fastest. For dense graphs it can be expected that the array methods will likewise be faster than the F-heap.

Priority Queue	Sparse	Middle	Dense
Array	$O(n^2)$	$O(n^2)$	$O(n^2)$
Aryptr	$O(n^2)$	$O(n^2)$	$O(n^2)$
Binary Heap	$O(n * \log n)$	$O(n * \log^2 n)$	$O(n^2 * \log n)$
Alpha Heap	$O(n * \log n)$	$O(n * \log^2 n / \log \log n)$	$O(n^2)$
F-heap	$O(n * \log n)$	$O(n * \log n)$	$O(n^2)$
Best	Binary Heap	F-Heap	Aryptr

Figure 4.1 The time bounds of the priority queues for graphs of different edge densities.

5. Graph Decisions

5.1 Graph Representation

To represent the graph in the computer's memory an adjacency list is used. This is an array of linked lists indexed by node number. For each node adjacent to a node, x the linked list pointed to by entry x of the array contains a record for that node. In this way it is possible to visit every node adjacent from some node by just traversing the corresponding list and furthermore this can be done in the least order of time possible, each access to a node being O(1). The construction of this list is also very efficient, taking just O(m) time. If instead a cost array, c is used where c[x,y] is the cost of the edge from node x to node y then $O(n^2)$ time would be needed to initialise it and O(n) time would be needed to visit every node adjacent from some node, x by scanning row x of the array.

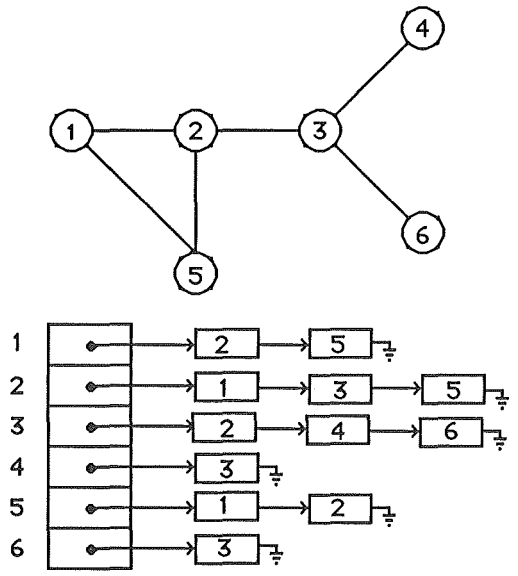


Figure 5.1 A graph and its adjacency list representation.

5.2 Worst Case Graph

In order to ensure the maximum possible number of decreasekey operations and worst case time for a decreasekey operation on each data structure it was necessary to find a graph cost function such that the tentative distance to each node as found by Dijkstra's algorithm would at each stage decrease and would become less than the minimum known distance still recorded in the data structure. Such a graph function was found by the author. An example of a graph of 5 nodes with this cost function is given in figure 5.2.

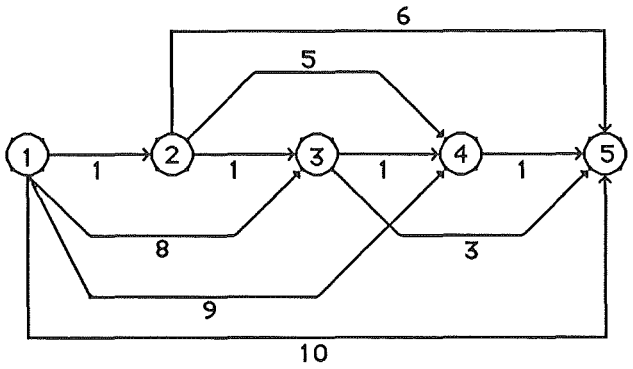


Figure 5.2 A worst case graph for n = 5

In this example it is assumed that Dijkstra's algorithm scans all the edges from a node in order of decreasing number of the adjacent node that is right to left in figure 5.2. Initially nodes 2 to 5 are inserted into the priority queue with cost infinity and node 1 is inserted with cost 0. Next assuming that node 1 is the source node the following is a trace of what happens when Dijkstra's algorithm is applied:

Current Node	Adjacent Node	Old Key	New Key	
-	1	-	0	<-- deletemin
1	5	∞	10	
1	4	∞	9	
1	3	∞	8	
1	2	∞	1	<-- deletemin
2	5	10	7	
2	4	9	6	
2	3	8	2	<-- deletemin
3	5	7	5	
3	4	6	3	<-- deletemin
4	5	5	4	<-- deletemin

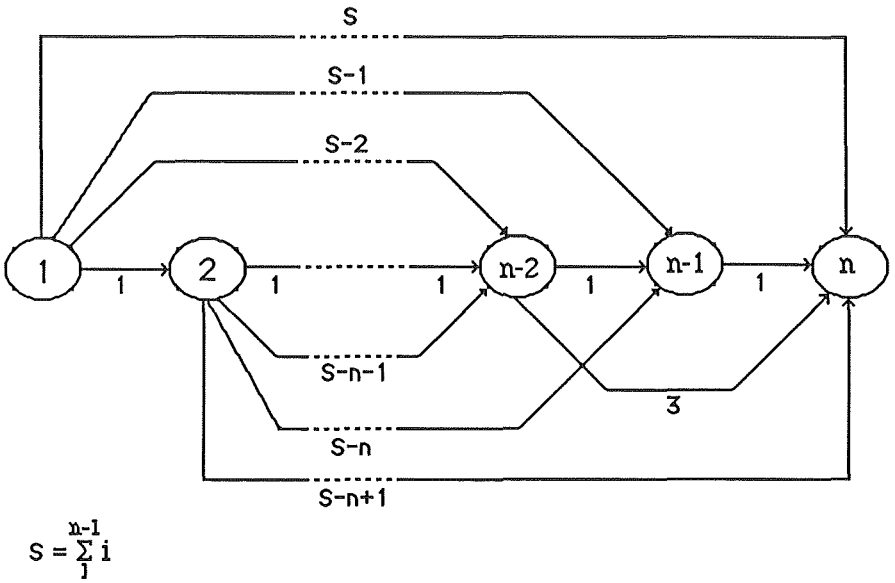
Figure 5.3 An example of running Dijkstra's algorithm for a worst case graph of 5 nodes.

Notice that each time a key value is updated by a decreasekey operation it changes to a value less than the current minimum, so that within the heap structures the associated node must be repositioned at the top of the heap which results in the worst possible time performance.

Generally a graph of n nodes with the cost function shown in figure 5.4 will produce the worst possible behaviour of Dijkstra's algorithm using any of the priority queues provided that the source is node 1.

$$cost(p,q) = \begin{cases} 1 & , q = p + 1 \\ \left(\sum_{i=1}^{n-p} i\right) - n + q & , p < q \\ \text{any } k \geq 0 & , p \geq q \end{cases}$$

Figure 5.4 The worst case cost function.



$$S = \sum_{1}^{n-1} i$$

Figure 5.5 A graph with the worst case cost function.

As is shown in figure 5.5 a worst case graph with n nodes and n * (n - 1) / 2 edges is formed by the algorithm described in the following pseudocode :

```
nextcost ← 2
For each node i from 1 to n-1 :
        cost( i, i+1 ) ← 1
For each node i from n-2 downto 1 :
        nextcost ← nextcost + 1
        For each node j from  i+2 to n :
                cost( i, j ) ← nextcost
                nextcost ← nextcost + 1
```

From the edges given to the graph by this algorithm, m needed to be chosen. The n-1 '1' edges between adjacent nodes were left to ensure that the nodes were deleted from the priority queue in the correct order. From the other nodes, m - n + 1 edges had to be chosen. Since this choice was unimportant, in that any choice would yield a graph which would result in m worst case decreasekey operations, the choice was made deterministically. This was done by adding the '1' edges and then spreading the remaining edges evenly over the remaining nodes. To accomplish this spreading of edges, the average p, of edges left per node was found and was used to find k = n - p - 2, the number of the node after which less than p more edges can be added because of a lack of destination nodes. All corresponding edges were then added to the nodes from k+1 to n-2 and the average p, of edges left over the remaining k nodes, was calculated. For each of the nodes i from k down to 1, the p shortest edges of the worst case graph, with the node i as source, were added until the graph contained m edges. In this way a worst case graph of n nodes and m edges was created in O(m) time.

6. Implementation

The programs needed for this project were written in Pascal with each priority queue a separately compiled unit and were executed on the Prime750 of the Computer Centre at the University of Canterbury. The measurements made were of the time for Dijkstra's algorithm to run and of the number of comparisons between keys and accesses of keys. The latter is important because although in the case of Dijkstra's algorithm the keys are simply numbers, the priority queues could be used for algorithms where the keys are objects for which comparisons and/or accesses require a great deal more time, for example they might be large records. Due to limitations of memory it was not possible to generate graphs of more than 2500 nodes. A graph of 2500 nodes is the largest and hence most interesting for which results are given in the following section.

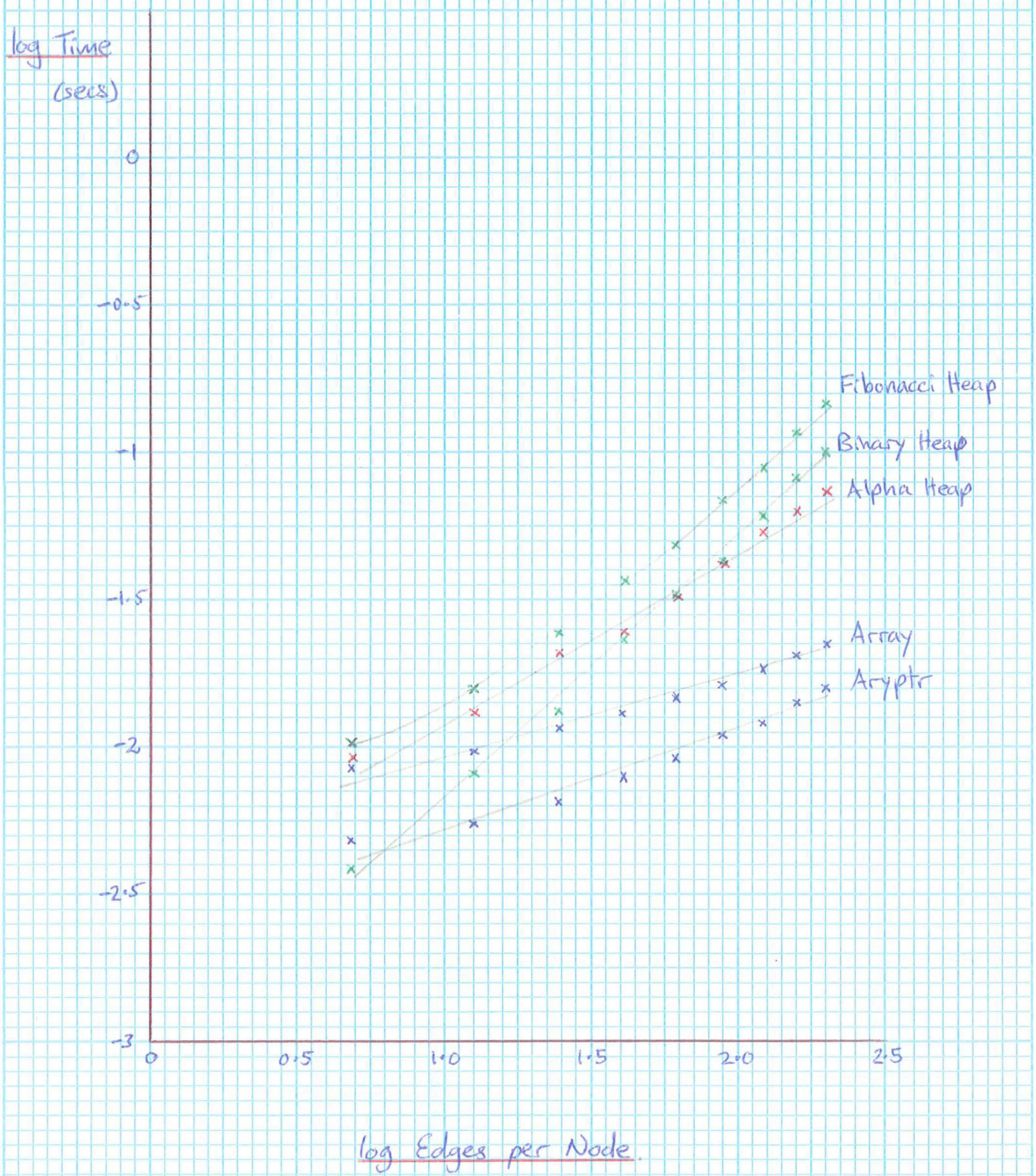
7. Results

All the results given in this report are averages over 10 runs. Some of the results are tabulated in the appendix. If necessary, further details may be obtained from the author.

7.1 Results Shown In Graph 1

Graph 1 shows the time taken by the five priority queues when Dijkstra's algorithm was run on worst case graphs of 100 nodes with edges per node varying from 2 to 10. The array priority queues clearly become faster than the others. This is due to the low overhead in these methods compared with that of the other three. Of the array methods aryptr is faster, even though it has a little more overhead, since the n deletemins only require half as many operations (see section 3.2). Although fastest here, the results for a worst case graph of 500 nodes shown on page A-12 of the appendix demonstrate that the arrays, as expected, eventually became much slower than the other methods. Consequently the array priority queues were not considered beyond these two runs.

Graph 1. Time vs Edges per Node for n=100.

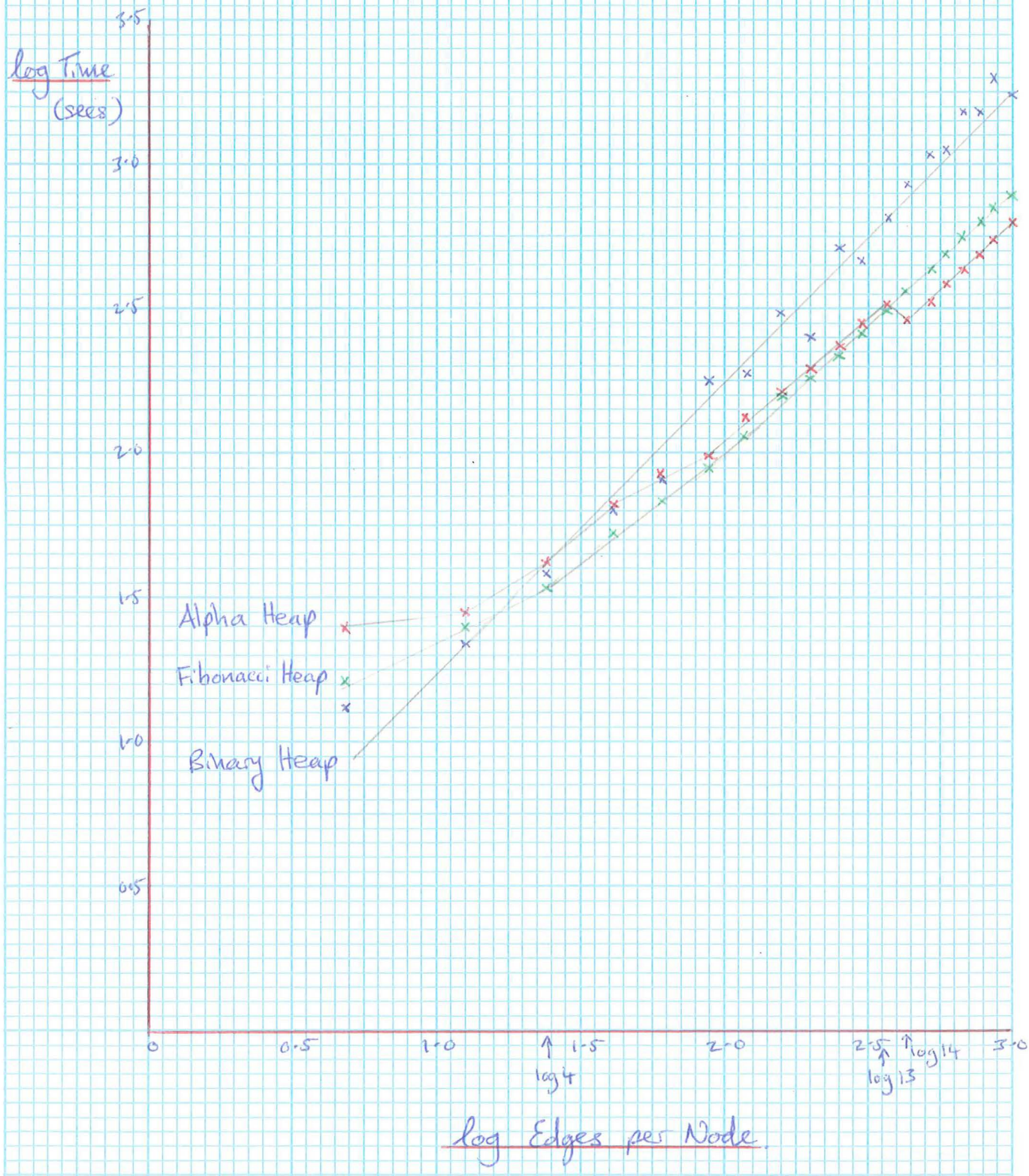


7.2 Results Shown In Graph 2

Graph 2 shows the time taken by Dijkstra's algorithm using the priority queues on a graph of 2500 nodes with edges per node varying from 2 to 20. From edges per node = 4 to 13 the F-heap is the fastest of the priority queues. As $m / n = \log_2(2500) \approx 11$, this range agrees with the best value of m for an F-heap predicted from the theory. For m/n = 14 the alpha heap becomes the fastest of the priority queues because of a sudden drop in the time taken from m/n = 13 to m/n = 14. This drop is explained by a decrease in the depth of the alpha heap; at m/n = 13, $\alpha = 13$ so the depth is $\lceil \log_\alpha(n(\alpha - 1) + 1) \rceil = 5$ while at m/n = 14, $\alpha =$ 14 so the depth is $\lceil \log_\alpha(n(\alpha - 1) + 1) \rceil = 4$. With a lower depth the alpha heap requires less time to swap records from the top to the bottom and vice versa in deletemin and decreasekey operations, respectively. Nevertheless swapping nodes toward the bottom of the heap in a deletemin operation requires requires scanning α records for the one of minimum key at each swap. Hence apart from drops in the time required, due to the depth decreasing, the time taken generally increases as α increases. There will be similar drops in the time taken when the heap depth decreases from 4 to 3 and from 3 to 2, these will occur from branching factors of $\alpha = 50$ to $\alpha = 51$ and $\alpha = 2498$ to $\alpha = 2499$. Although there are drops in the heap depth for regions shown in the graph other than the one cited above, these earlier drops had less of an effect on the time taken since the decrease was less in proportion to the initial depth.

The binary heap is initially, for a sparse graph, the fastest of the methods. This is expected from the theory.

Graph 2. Time vs. Edges per Node for $n = 2500$.

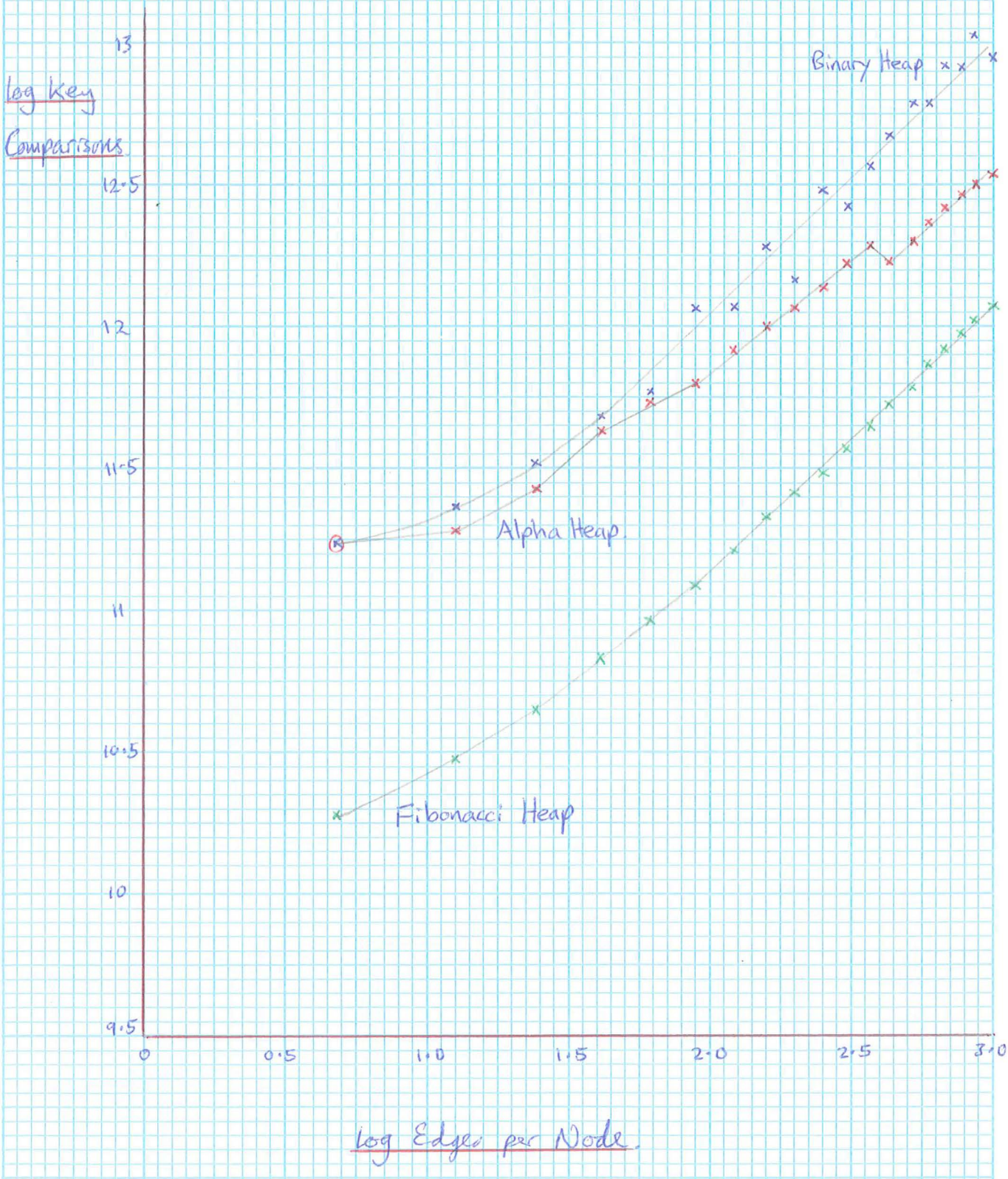


7.3 Results Shown In Graph 3 and 4

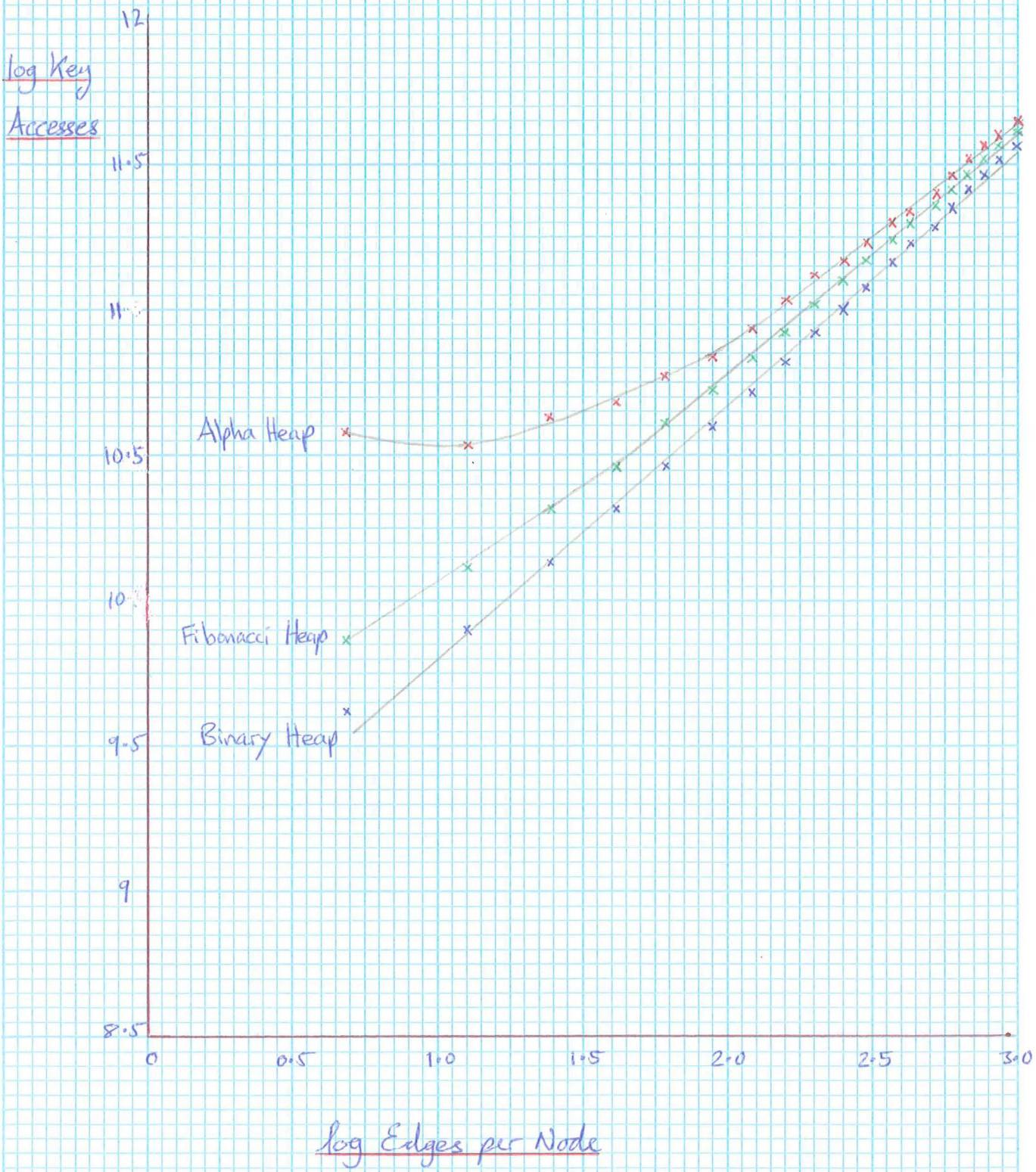
Graph 3 shows the number of key comparisons made by each of the three heap priority queues used in Dijkstra's algorithm on a worst case graph of 2500 nodes. Clearly the F-heap makes far less comparisons between keys than the other priority queues. From the results listed on page A-5 of the appendix it can be found that the Fibonacci heap varies between making 1.2 to 2.6 less key comparisons than its nearest rival the alpha heap. The sudden drop in the alpha heap graph from $\alpha = 13$ to 14 can be explained as in 7.1 by a decrease in the depth of the heap.

Graph 4 shows the number of key accesses made by each of the heap priority queues for worst case graphs of 2500. These show that there is little difference between the three methods in the number of accesses. Although the binary heap makes the fewest accesses this is not a significant advantage as it only makes a few thousand less than the F-heap which makes a few hundred thousand less key comparisons than the binary heap. Key comparisons are likely to take much greater time than accesses in any case. The graphs show that the binary heap makes the fewest accesses followed by the F-heap and the alpha heap makes the most accesses.

Graph 3 Number of Key Comparisons vs Edges per Node for n = 2500.



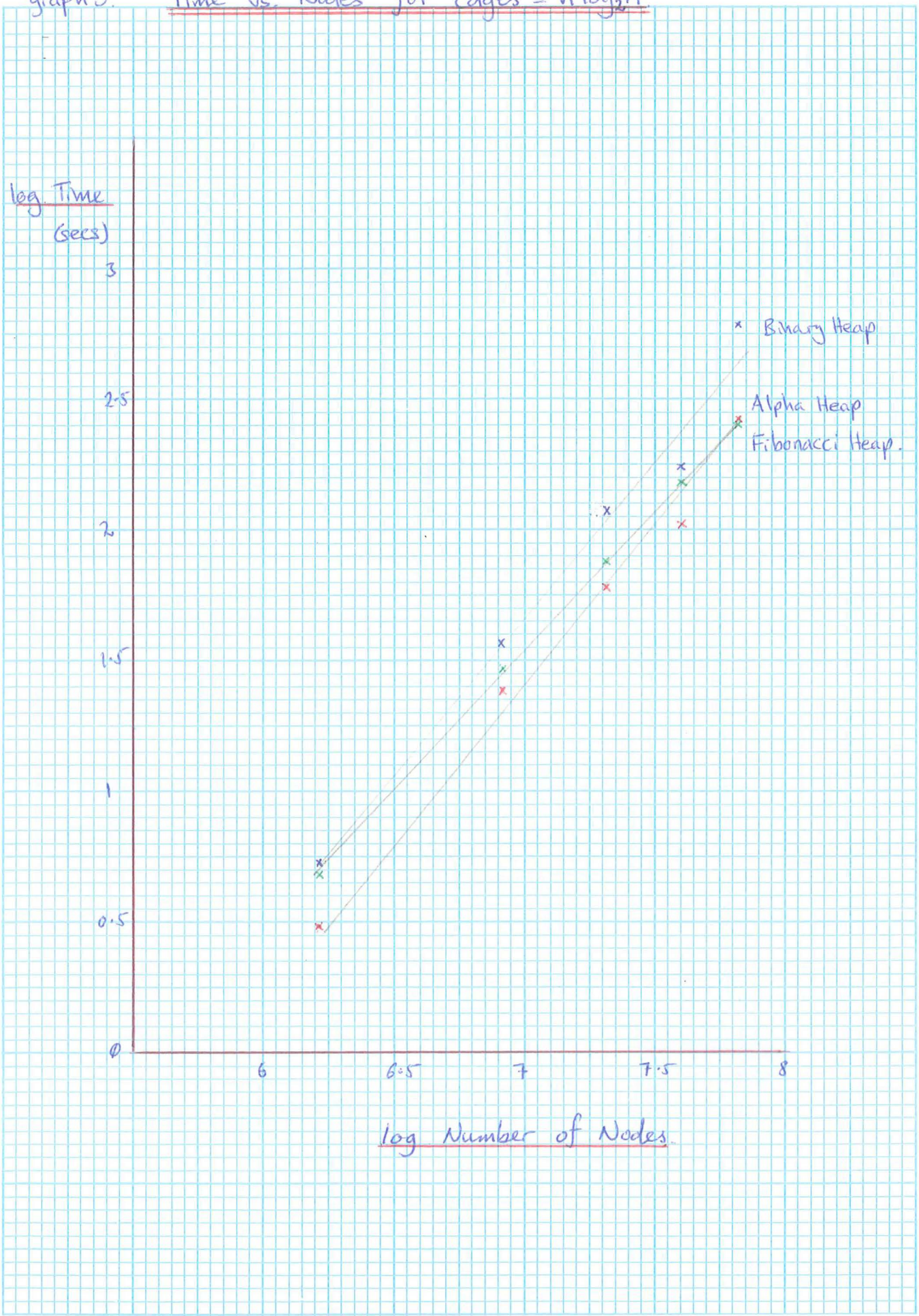
Graph 4. Number of Key Accesses vs Edges per Node for $n = 2500$



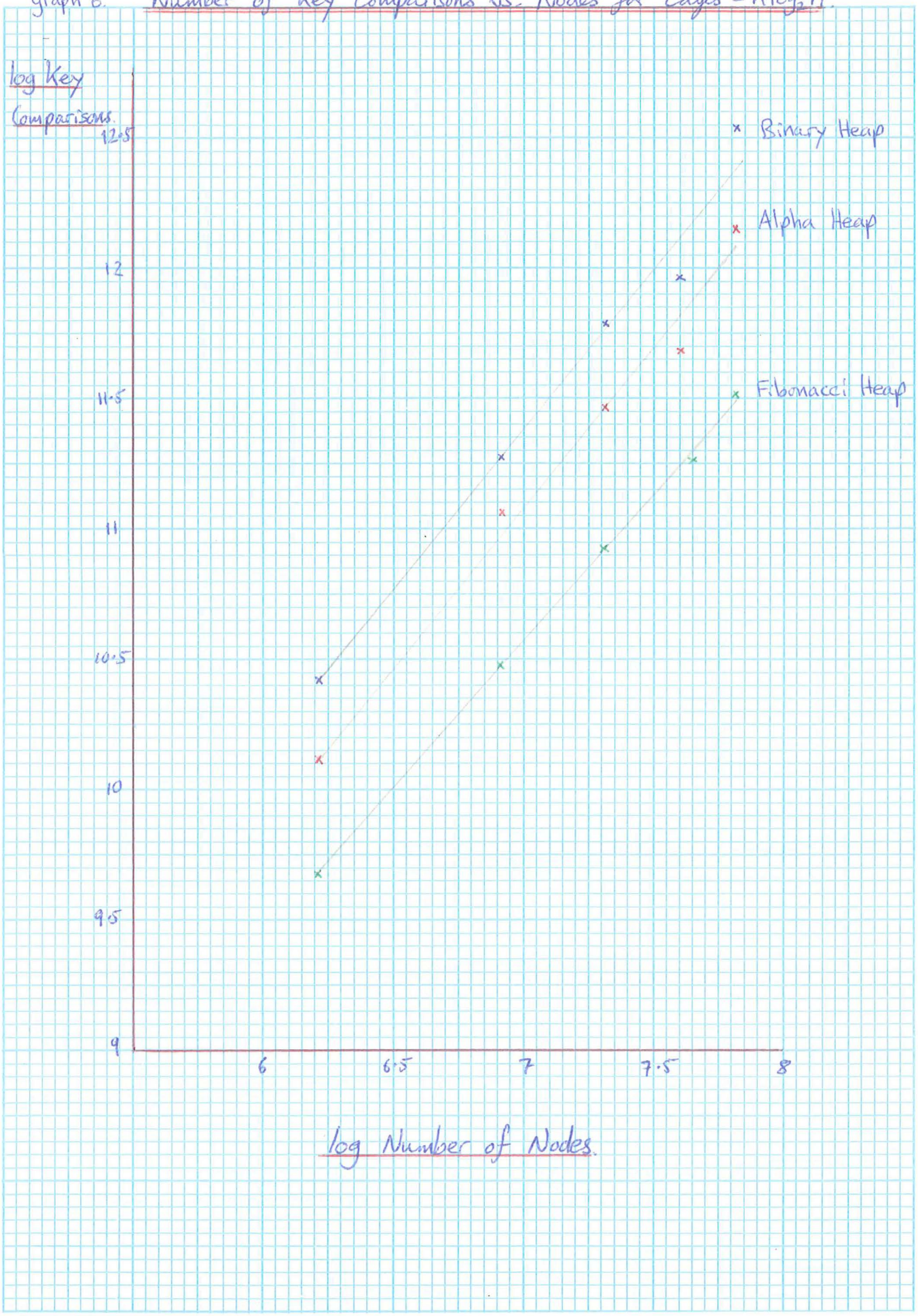
7.4 Results Shown In Graphs 5 to 7

Graphs 5 through 7 show the performance of the three heap priority queues for varying numbers of nodes with the number of edges kept constant at n * log n. Graph 5 shows that the F-heap only just gives the best time for a worst case graph of 2500 nodes. Graph 6 shows that the F-heap makes significantly less key comparisons than the other methods while graph 7 shows that there is little difference in the number of key accesses made by each priority queue, relative to the other considerations as explained in section 7.2 above.
These three graphs demonstrate that if Dijkstra's algorithm is applied to a worst case graph of n nodes and n * log n edges, the alpha heap used as a priority queue gives the best time performance for values of n from about 500 to 2000, however from n = 2500 the Fibonacci Heap is faster.

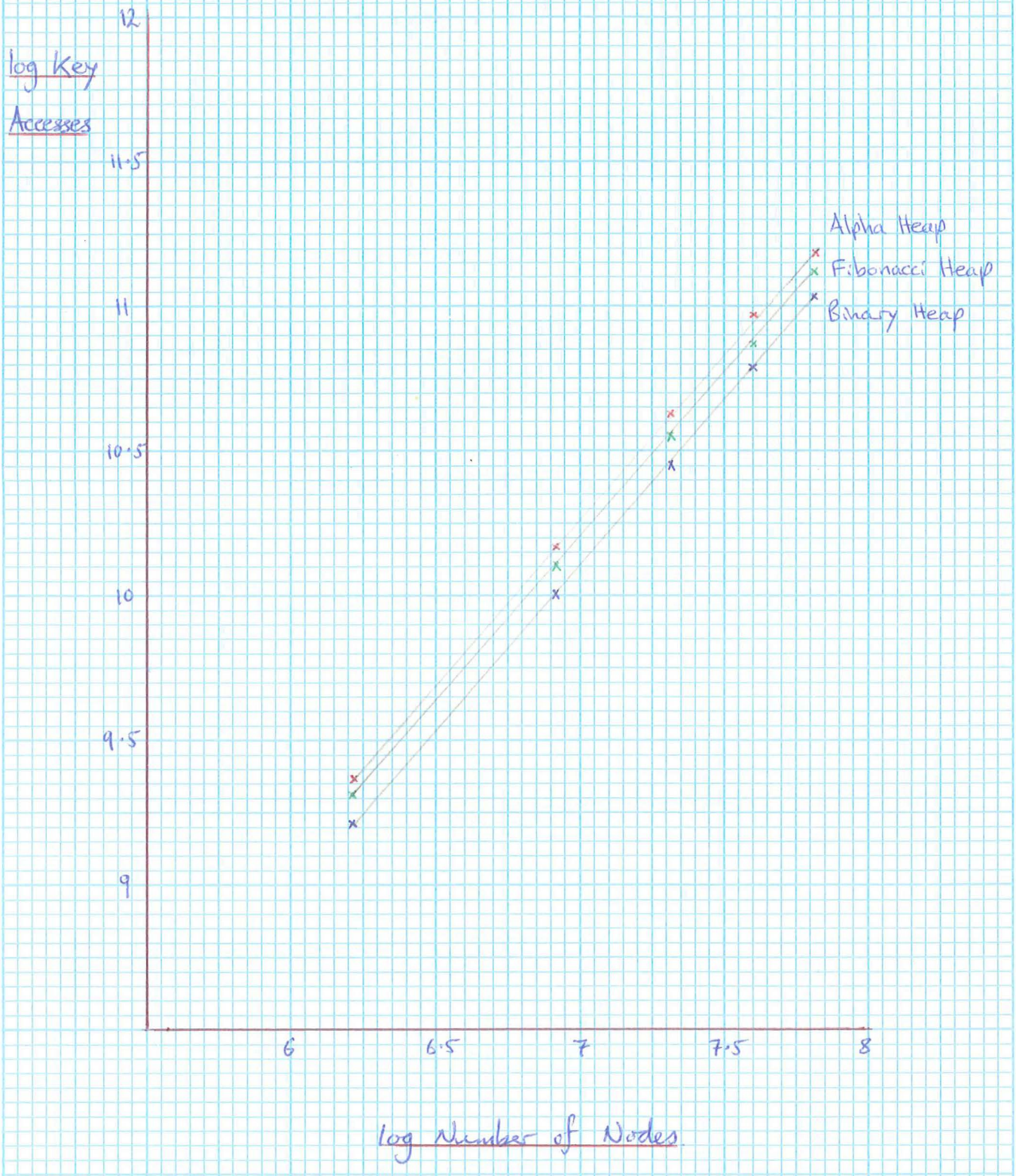
Graph 5. Time vs. Nodes for Edges = $n \log_2 n$.



Graph 6. Number of Key Comparisons vs. Nodes for Edges = $n \log_2 n$.



Graph 7. Number of Key Accesses vs. Nodes for Edges = $n \log_2 n$.



7.5 Summary of Results

Figure 7.1 shows the best priority queue for varying numbers of nodes and varying densities of edges for Dijkstra's Shortest Path algorithm applied to a worst case graph. The last column indicates the well known result that arrays are fastest when used for dense graphs.

Nodes	Sparse Graph	Middle Graph	Dense Graph
500	Binary Heap	Alpha Heap	Aryptr
1000	Binary Heap	Alpha Heap	Aryptr
1500	Binary Heap	Alpha Heap	Aryptr
2000	Binary Heap	Alpha Heap	Aryptr
2500	Binary Heap	Fibonacci Heap	Aryptr

Figure 7.1 The fastest priority queues for a worst case graph.

These results apply to integer keys. If however the keys are very large and hence take much longer than do integers to be compared then the Fibonacci heap is likely to be significantly faster than the other heaps in areas where it is not shown to be in figure 7.1 since the Fibonacci heap was found to make greatly fewer key comparisons for both sparse and middle type graphs.

8. Conclusion

The project was successful in showing Fibonacci heaps to be faster than other commonly used priority queues when used by Dijkstra's Single Source Shortest Path algorithm on sufficiently large worst case graphs. It was also found that the F-heap makes significantly less key comparisons than the other methods, which consequently would make it even faster, compared with the other priority queues, for sufficiently large keys.

References

[FredTarj84] M. L. Fredman, R. E. Tarjan; "Fibonacci Heaps and Their Uses In Improved Network Optimization Algorithms", IEEE Computer Society 25th Conference on Foundations of Computer Science, Oct 1984, 338 - 346.

[HoroSahn84] E. Horowitz, S. Sahni; "Fundamentals of Data Structures in Pascal", Computer Science Press, 1984, 349 - 350.

[AHU74] A. V. Aho, J. E. Hopcroft, J. D. Ullman; "The Design and Analysis of Computer Algorithms", Addison-Wesley, Reading, Massachusetts, 1974, 207 - 209.

[AHU83] A. V. Aho, J. E. Hopcroft, J. D. Ullman; "Data Structures and Algorithms", Addison-Wesley, Reading, Massachusetts, 1983, 205 - 207.

Appendix

Time in CPU Seconds for Nodes = 2500, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	3.063	4.053	3.352
4.	4.846	5.071	4.664
6.	6.778	6.933	6.267
8.	9.464	9.730	8.321
10.	9.028	11.071	9.886
12.	10.359	14.412	11.555
14.	12.111	18.727	11.730
16.	13.842	21.048	13.280
18.	15.697	24.042	14.781
20.	17.212	25.607	16.142

Time in CPU Seconds for Nodes = 2000, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	2.372	3.135	2.681
4.	3.769	4.044	3.787
6.	5.384	5.550	5.079
8.	7.631	6.537	6.441
10.	9.733	7.779	7.801
12.	11.687	9.192	9.095
14.	14.267	9.251	10.618
16.	15.997	10.554	12.006
18.	18.335	11.664	13.416
20.	20.333	12.923	14.686

Time in CPU Seconds for Nodes = 1500, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	1.756	2.322	2.008
4.	2.824	3.048	2.825
6.	3.974	3.814	3.847
8.	5.967	4.826	4.887
10.	7.375	5.823	5.985
12.	8.336	6.016	6.951
14.	10.169	6.981	8.110
16.	10.698	7.870	9.190
18.	13.387	8.762	10.318
20.	15.478	9.589	11.248

Time in CPU Seconds for Nodes = 1000, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	1.110	1.462	1.306
4.	1.792	1.922	1.854
6.	2.601	2.514	2.516
8.	3.349	3.147	3.176
10.	4.617	3.421	3.884
12.	5.434	4.028	4.526
14.	6.780	4.607	5.261
16.	7.691	5.170	5.963
18.	8.310	5.737	6.689
20.	9.018	6.270	7.304

Time in CPU Seconds for Nodes = 500, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	1.110	1.462	1.306
4.	1.792	1.922	1.854
6.	2.601	2.514	2.516
8.	3.349	3.147	3.176
10.	4.617	3.421	3.884
12.	5.434	4.028	4.526
14.	6.780	4.607	5.261
16.	7.691	5.170	5.963
18.	8.310	5.737	6.689
20.	9.018	6.270	7.304

Number of Key Comnparisons for Nodes = 2500, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	76346.	76346.	29264.
4.	100704.	92834.	42118.
6.	128692.	125692.	57539.
8.	173479.	174698.	148873.
10.	83169.	191589.	173483.
12.	97677.	247177.	203126.
14.	115130.	318590.	205578.
16.	132043.	354806.	232122.
18.	150618.	403164.	257485.
20.	165362.	421731.	279207.

Number of Key Comnparisons for Nodes = 2000, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	58629.	58629.	23344.
4.	77310.	74028.	33627.
6.	100721.	100274.	45969.
8.	136179.	116862.	58758.
10.	168806.	135621.	72126.
12.	199188.	160928.	84658.
14.	241360.	161265.	98930.
16.	265458.	183406.	112736.
18.	305110.	200768.	126554.
20.	335775.	220702.	138928.

Number of Key Comnparisons for Nodes = 1500, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	42889.	42889.	17022.
4.	57344.	54460.	24734.
6.	74328.	68051.	33992.
8.	105147.	84527.	43583.
10.	126929.	100976.	53613.
12.	139324.	103750.	63009.
14.	171388.	120323.	73713.
16.	174651.	134631.	84066.
18.	220162.	147742.	94431.
20.	255624.	159917.	103710.

Number of Key Comnparisons for Nodes = 1000, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	26556.	26556.	11164.
4.	35535.	33896.	16305.
6.	47284.	44033.	22482.
8.	57444.	53765.	28874.
10.	78554.	58377.	35559.
12.	90631.	68357.	41828.
14.	112690.	77503.	48965.
16.	126132.	85788.	55867.
18.	134081.	92883.	62771.
20.	144448.	98878.	68971.

Number of Key Comnparisons for Nodes = 500, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	11839.	11839.	5326.
4.	16353.	15831.	7898.
6.	22083.	20054.	10988.
8.	26999.	23072.	14181.
10.	35298.	27657.	17531.
12.	41796.	31177.	20677.
14.	49792.	34604.	24228.
16.	53991.	37473.	27675.
18.	59603.	41930.	31124.
20.	66643.	46069.	34236.

Number of Key Accesses for Nodes = 2500, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	15001.	39158.	19063.
4.	25001.	41154.	30079.
6.	35001.	47639.	40464.
8.	45001.	56382.	50373.
10.	55001.	67263.	60639.
12.	65001.	75419.	70887.
14.	75001.	84429.	80796.
16.	85001.	94463.	90532.
18.	95001.	104362.	100848.
20.	105001.	114192.	111025.

Number of Key Accesses for Nodes = 2000, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	12001.	30663.	15251.
4.	20001.	31859.	24063.
6.	28001.	38168.	32371.
8.	36001.	45098.	40298.
10.	44001.	52113.	48512.
12.	52001.	60441.	56708.
14.	60001.	67535.	64637.
16.	68001.	75490.	72430.
18.	76001.	83316.	80680.
20.	84001.	91148.	88820.

Number of Key Accesses for Nodes = 1500, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	9001.	22439.	11438.
4.	15001.	23692.	18048.
6.	21001.	27673.	24279.
8.	27001.	34034.	30224.
10.	33001.	39224.	36384.
12.	39001.	44589.	42531.
14.	45001.	50580.	48476.
16.	51001.	56473.	54324.
18.	57001.	62331.	60512.
20.	63001.	68292.	66619.

Number of Key Accesses for Nodes = 1000, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	6001.	14329.	7626.
4.	10001.	15129.	12033.
6.	14001.	18375.	16185.
8.	18001.	22038.	20151.
10.	22001.	25662.	24258.
12.	26001.	29679.	28355.
14.	30001.	33558.	32318.
16.	34001.	37472.	36220.
18.	38001.	41573.	40343.
20.	42001.	45753.	44415.

Number of Key Accesses for Nodes = 500, Edges per Node = 2 .. 20.

Edges/Node	Binary Heap	Alpha Heap	Fibonacci Heap
2.	3001.	6663.	3813
4.	5001.	7437.	6017
6.	7001.	9412.	8093
8.	9001.	10766.	10074
10.	11001.	12768.	12129
12.	13001.	14752.	14178
14.	15001.	16849.	16159
16.	17001.	18494.	18114
18.	19001.	20483.	20176
20.	21001.	22467.	22210

Time in CPU Seconds for Nodes = 100, Edges per Node = 2 .. 10.

Edges/Node	Array	Aryptr	Binary Heap	Alpha Heap	Fibonacci Heap
2.	0.126	0.099	0.090	0.130	0.137
3.	0.133	0.104	0.124	0.152	0.166
4.	0.143	0.112	0.153	0.186	0.200
5.	0.151	0.123	0.195	0.198	0.238
6.	0.159	0.131	0.228	0.225	0.270
7.	0.168	0.141	0.253	0.252	0.314
8.	0.176	0.147	0.297	0.280	0.350
9.	0.184	0.157	0.341	0.302	0.394
10.	0.193	0.165	0.368	0.322	0.431

Number of Key Comparisons for Nodes = 100, Edges per Node = 2 .. 10.

Edges/Node	Array	Aryptr	Binary Heap	Alpha Heap	Fibonacci Heap
2.	9900.	4950.	1698.	1698.	932.
3.	9900.	4950.	2118.	1995.	1191.
4.	9900.	4950.	2523.	2487.	1448.
5.	9900.	4950.	3069.	2626.	1797.
6.	9900.	4950.	3489.	2914.	2067.
7.	9900.	4950.	3764.	3231.	2392.
8.	9900.	4950.	4375.	3597.	2709.
9.	9900.	4950.	5082.	3882.	3089.
10.	9900.	4950.	5401.	4101.	3375.

Number of Key Accesses for Nodes = 100, Edges per Node = 2 .. 10.

Edges/Node	Array	Aryptr	Binary Heap	Alpha Heap	Fibonacci Heap
2.	899.	795.	601.	1096.	811.
3.	1099.	1042.	801.	1175.	1011.
4.	1299.	1289.	1001.	1374.	1226.
5.	1499.	1535.	1201.	1514.	1411.
6.	1699.	1782.	1401.	1740.	1632.
7.	1899.	2029.	1601.	1884.	1825.
8.	2099.	2277.	1801.	2083.	2025.
9.	2299.	2525.	2001.	2284.	2221.
10.	2499.	2774.	2201.	2483.	2434.

Time in CPU Seconds for Edges = n * log n.

Nodes	Edges	Binary Heap	Alpha Heap	Fibonacci Heap
500.	4483.	2.072	1.631	1.971
1000.	9966.	4.795	4.013	4.341
1500.	15826.	7.908	5.903	6.552
2000.	21932.	9.403	7.559	8.838
2500.	28219.	16.208	11.238	11.057

Number of Key Comparisons for Edges = n * log n.

Nodes	Edges	Binary Heap	Alpha Heap	Fibonacci Heap
500.	4483.	33619.	24518.	16056.
1000.	9966.	78429.	63806.	35456.
1500.	15826.	131886.	94432.	56005.
2000.	21932.	157006.	118092.	77742.
2500.	28219.	280314.	189440.	99758.

Number of Key Accesses for Edges = n * log n.

Nodes	Edges	Binary Heap	Alpha Heap	Fibonacci Heap
500.	4483.	9967.	11766.	11078.
1000.	9966.	21933.	26142.	24284.
1500.	15826.	34653.	41324.	38216.
2000.	21932.	47865.	58206.	52647.
2500.	28219.	61439.	71792.	67561.

Results for n = 500, m = 1000.

Priority Queue	Time (secs)	Key Comparisons	Key Accesses
Array	2.669	249500.	4499.
Aryptr	1.651	124750.	3995.
Binary Heap	0.534	11839.	3001.
Alpha Heap	0.800	11839.	6663.
Fibonacci Heap	0.730	5326.	4061.